

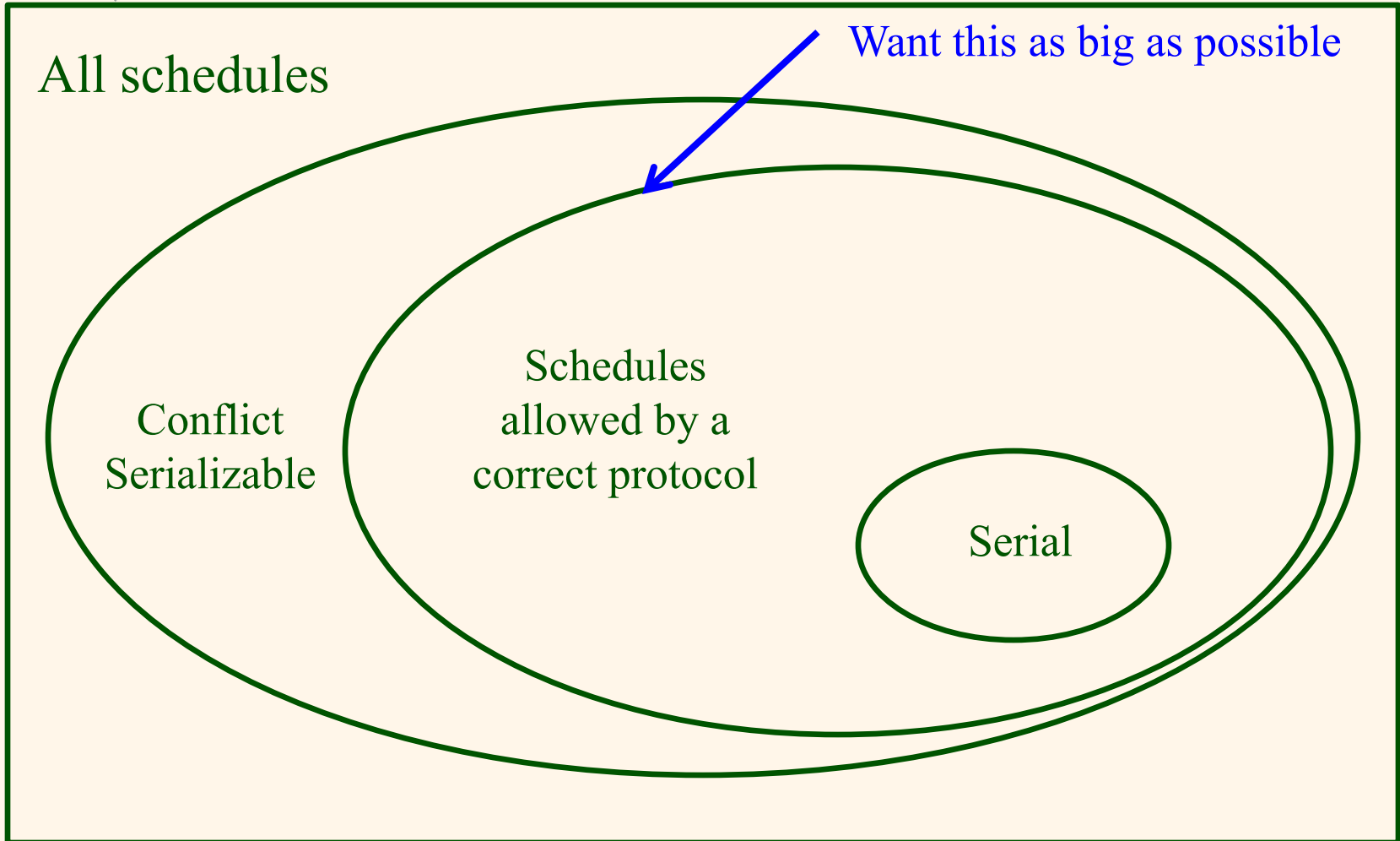
Concurrency Control - Two-Phase Locking




Last time

- ❖ Conflict serializability
- ❖ Protocols to enforce it


Big Picture





Locking-Based Protocols

- ❖ First family of protocols – based on idea of locks
- ❖ Before any read or write, a transaction must request a lock on an object
 - A "permission to operate" on this object
- ❖ Locks are managed centrally by the DBMS lock manager



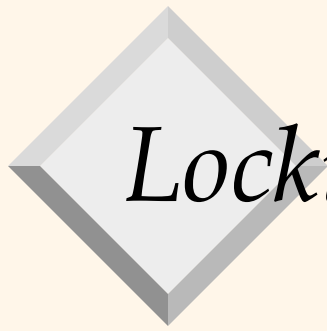
Locking-Based Protocols

- ❖ Locks can last only as long as a single operation (short locks)
 - Good idea to use those anyway – prevent two transactions from writing to the same object literally at the same time
- ❖ But can also allow transactions to hold on to them for longer



A simple locking scheme

- ❖ Before execution, each transaction locks entire database
- ❖ After it commits, releases lock for next transaction




Locking objects

- ❖ For greater concurrency, allow locks on individual DB objects
- ❖ Locking scheme #2:
 - Transaction comes into system
 - Requests locks on **all objects it needs to access**
 - When obtains locks, proceeds
 - On commit, releases all locks
- ❖ Why is this better already?
 - And does it still enforce conflict serializability?



Another refinement

- ❖ If a transaction T has a lock on A, no-one else can operate on A until it finishes
- ❖ But suppose T is **only reading A**
 - And T' also only wants to read A
 - Can't we allow them both in?
- ❖ Realistic scenario: in a lot of DB workloads, reads much more frequent than writes...



Kinds of locks

- ❖ Idea: refine locks into two kinds:
 - read locks – can be shared
 - write locks – must be exclusive
- ❖ Transaction explicitly asks system for either read or write lock
 - When can system grant a read lock?
 - When can system grant a write lock?



Locking Scheme #3

- ❖ Allows us to have a new locking scheme:
 - Transaction requests read/write locks on all the objects it will need to access
 - ◆ If needs to both read and write to object, must request write lock
 - Waits until it has all of them
 - Runs
 - On commit, releases all locks
- ❖ Does it still guarantee conflict serializability?



More refinements

- ❖ Can we allow transactions to release some locks before commit?
 - If transaction is done with the object, should be fine
 - Problem: some dirty reads now possible
 - **W1(A) R2(A) Abort1**
 - Need to abort 2 as well
 - A **cascading abort**
- ❖ But would guarantee conflict serializability
 - Just not the ACA or strictness properties we discussed



More refinements

- ❖ Can we allow transactions to acquire locks "as needed" rather than all at once?
 - Yes, but **deadlock** can happen!
 - Transactions 1 and 2 both need write locks on A and B
 - Transaction 1 gets lock on A
 - Transaction 2 gets lock on B
 - Neither can proceed!
 - Need some deadlock handling algorithm
- ❖ But would still be correct (guarantee conflict serializability)



More refinements

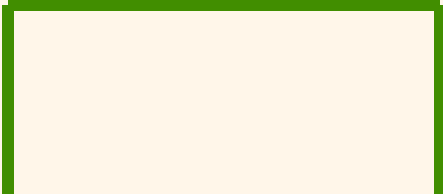
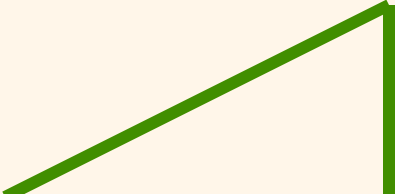
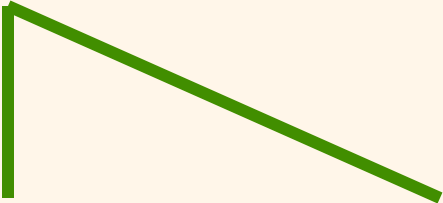
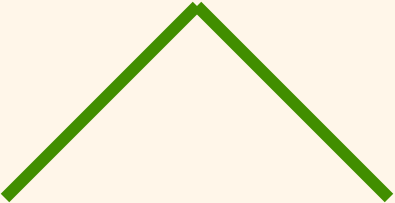
- ❖ Can even do both:
 - acquire locks as needed
 - release when no longer needed
- ❖ But to guarantee conflict-serializability, need to have two separate phases:
 - Phase 1: only acquire locks
 - Phase 2: only release locks
- ❖ Two-phase locking (2PL)



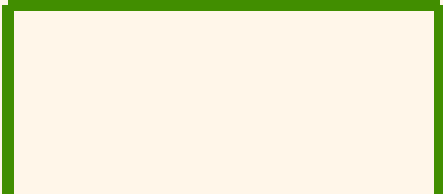
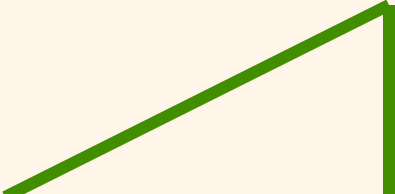
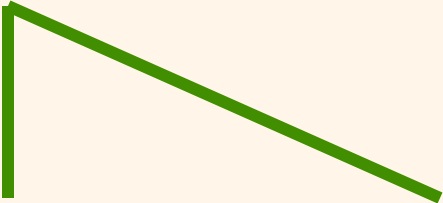
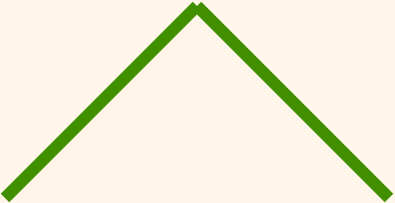
Protocols

- ❖ Each transaction locks **entire DB**
- ❖ Each transaction only **locks objects it needs**
- ❖ Each transaction specifies whether it needs **read or write locks**
- ❖ Transaction can release locks before commit if it's done
 - Problem: cascading aborts
- ❖ Transaction can acquire locks as needed
 - Problem: deadlocks

Locking options

| Acquire Release | At the start | As needed |
|--------------------|---|--|
| Upon Commit |  |  |
| Early |  |  |

2PL variants

| <div>Conservative</div> <div>Strict</div> | Yes | No |
|---|---|--|
| |  |  |
| Yes | | |
| No |  |  |



Pros and cons

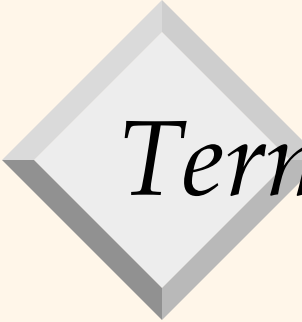
- ❖ Conservative (get all locks at start)
 - + : no deadlocks
 - - : need to know all objects up front
 - - : less concurrency

- ❖ Strict (hold locks until commit)
 - +: no cascading aborts
 - - : less concurrency



Summary of Alternatives

- ❖ Conservative Strict 2PL
 - No deadlocks, no cascading aborts, no need to know when to release locks
 - **But** need to know objects a priori
- ❖ Conservative 2PL
 - No deadlocks, more concurrency than Conservative Strict 2PL
 - **But** need to know objects a priori, when to release locks, cascading aborts
- ❖ Strict 2PL
 - No cascading aborts, no need to know objects a priori or when to release locks, more concurrency than Conservative Strict 2PL
 - **But** deadlocks
- ❖ "Plain" 2PL (non-Strict, non-Conservative)
 - Most concurrency, no need to know object a priori
 - **But** need to know when to release locks, cascading aborts, deadlocks



Terminology warning

- ❖ 2PL (Two Phase Locking) may refer to:
 - The whole family of techniques, OR
 - The non-Strict, non-Conservative variant (I will sometimes call this **plain 2PL** to disambiguate)



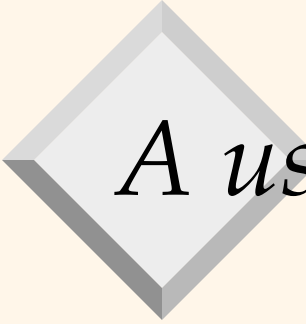
Let's prove that 2PL is OK!

- ❖ Let's prove that (non-strict, non-conservative) 2PL enforces conflict serializability
- ❖ To show: suppose system uses 2PL and produces some schedule
 - The conflict graph for this schedule must be acyclic
- ❖ (See Bernstein's textbook, Section 3.3.)



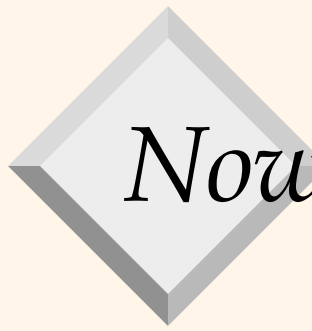
Warm-up: enhancing schedules with lock requests

- ❖ Given a schedule, suppose it was executed in a system running 2PL (don't know if it was Strict/Conservative)
- ❖ Think about possible places where lock and unlock could have happened
- ❖ E.g. W1(A)W1(B)
 - All possible places to put lock/unlock requests under 2PL?



A useful observation

- ❖ Under 2PL, a transaction cannot issue a release lock request *followed by* an acquire lock request (by definition of 2PL)



Now back to our proof

- ❖ By contradiction: suppose we have a schedule produced in a system using 2PL
- ❖ And the schedule is not conflict serializable
- ❖ What does that mean?



A useful lemma

- ❖ Consider the conflict graph for our schedule.
If there is a directed path from i to j , then i releases some lock before j acquires some lock
 - not necessarily on same object
- ❖ Let's prove that!
- ❖ Use induction on path length and our previous observation



A useful lemma

- ❖ Consider the conflict graph for our schedule.
If there is a directed path from i to j , then i releases some lock before j acquires some lock
 - not necessarily on same object
- ❖ Suppose now the graph has a cycle; how does it relate to the above?
- ❖ And to 2PL?